RiskRanker: Scalable and Accurate Zero-day Android Malware Detection

Michael Grace †, Yajin Zhou †, Qiang Zhang ‡, Shihong Zou ‡, Xuxian Jiang †*
†North Carolina State University ‡NQ Mobile Security Research Center
{mcgrace, yajin_zhou, xjiang4}@ncsu.edu {zhangqiang, zoushihong}@nq.com

ABSTRACT

Smartphone sales have recently experienced explosive growth. Their popularity also encourages malware authors to penetrate various mobile marketplaces with malicious applications (or apps). These malicious apps hide in the sheer number of other normal apps, which makes their detection challenging. Existing mobile anti-virus software are inadequate in their reactive nature by relying on known malware samples for signature extraction. In this paper, we propose a proactive scheme to spot zero-day Android malware. Without relying on malware samples and their signatures, our scheme is motivated to assess potential security risks posed by these untrusted apps. Specifically, we have developed an automated system called *RiskRanker* to scalably analyze whether a particular app exhibits dangerous behavior (e.g., launching a root exploit or sending background SMS messages). The output is then used to produce a prioritized list of reduced apps that merit further investigation. When applied to examine 118,318 total apps collected from various Android markets over September and October 2011, our system takes less than four days to process all of them and effectively reports 3281 risky apps. Among these reported apps, we successfully uncovered 718 malware samples (in 29 families) and 322 of them are zero-day (in 11 families). These results demonstrate the efficacy and scalability of RiskRanker to police Android markets of all stripes.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and protection – *Invasive software*

General Terms

Security

Keywords

Android, Malware, RiskRanker

*The names of the first two authors are in alphabetical order.

MobiSys'12, June 25–29, 2012, Low Wood Bay, Lake District, UK.

1. INTRODUCTION

In recent years, smartphones have experienced explosive growth. Gartner [13] reports that worldwide smartphone sales in the third quarter of 2011 reached 115 million units – an increase of 42 percent from the third quarter of the previous year. CNN [28] similarly shows that smartphone shipments have tripled in the past three years. Not surprisingly, multiple smartphone platforms are vying for dominance on these mobile devices. At present, Google's Android platform has overtaken Symbian and iOS to become the most popular smartphone platform, being installed on more than half (52.5%) of all smartphones shipped [13].

The availability of feature-rich applications (or simply apps) is one of the key selling points that these mobile platforms advertise. By making it convenient for app developers to develop and publish apps, and easy for users to locate and install these apps, platform providers hope to set up a positive feedback loop in which apps will further attract users to their platforms, which in turn drive developers to develop more apps. Various organizations, therefore, have created app stores to facilitate this process. Platform providers tend to offer official distribution services such as Google's Android Market[1] or Apple's App Store. Cellular carriers also provide their own markets and stores, such as AT&T's AppCenter. Moreover, there are third-party markets altogether, ranging from publishing giant Amazon's Appstore to small, specialty markets like Freeware Lovers.

As platforms become more popular, it seems inevitable that they begin to attract developers of a different kind: malware authors. Moreover, the central role these markets play makes it possible for a tremendous number of mobile devices to be compromised in a very short time. For instance, the `DroidDream` malware infected more than 260,000 devices within 48 hours, before Google removed the related malicious apps from the official Android Market [1]. In light of these threats, there is a pressing need for market curators to examine or vet apps before accepting them for publication.

Unfortunately, the sheer number of new apps uploaded into these markets makes such examination challenging. Using the official Android Market as an example, in the first half of 2011 alone, 223,613 new apps were published [7], which translates to an average of 1242 new apps each day. Examining such a large number of apps manually – in a timely fashion – is a daunting task. We could choose to deploy mobile anti-virus software to scan these uploaded apps before they are made available for download. However, the

[1]The official Android Market is now part of Google Play.

reactive nature of existing mobile anti-virus software makes it inadequate in identifying new or mutated malicious apps. Specifically, such software relies solely upon a priori knowledge of malware samples in order to extract and deploy signatures for subsequent detection. From another perspective, malware authors may produce new malware variants, or obfuscate existing ones, to evade detection. For instance, the `DroidKungFu` malware has at least 5 different variants; each variant was able to escape detection by existing anti-virus software when it was first reported.

In this paper, we propose a *proactive* scheme to spot zero-day Android malware by scalably and accurately sifting through the large number of untrusted apps in existing Android markets, including both official and alternative ones. Without relying on malware specimens (and their signatures), our scheme is motivated and accordingly designed to measure potential security risks posed by these untrusted apps. Specifically, we divide potential risks into three categories: *high-risk*, *medium-risk*, and *low-risk*. High-risk apps exploit platform-level software vulnerabilities to compromise the phone integrity without proper authorization from users. Medium-risk apps do not exploit software vulnerabilities, but can cause users financial loss or disclose their sensitive information. For example, these apps may illicitly subscribe to premium services unbeknownst to the user. Low-risk apps are similar, but milder; they may collect device-specific or generic, generally readily-available personal information.[2]

Based on this risk classification, we have accordingly developed an automated system called *RiskRanker* to assess the risks from existing (untrusted) apps for zero-day malware detection. The assessment performs a two-order risk analysis. In the first-order risk analysis, we aim to directly identify apps in high- and medium-risk categories. For example, if an app contains code designed to exploit platform-level vulnerabilities (Section 2), it will be flagged as a high-risk app. In the second-order risk analysis, we perform a further investigation to uncover suspicious app behavior. For example, some malicious apps may be designed to encrypt exploit code to evade our first-order analysis. With that in mind, we develop systematic ways to locate those apps and map them to corresponding risk categories. By focusing on these high- and medium-risk apps, we can substantially reduce the number of suspicious apps that require subsequent verification.

We have implemented a RiskRanker prototype and evaluated it using 118,318 apps (104,874 distinct apps)[3] collected over a two-month period, i.e., September and October 2011. We deploy our system and run it in parallel on a local cluster of five machines. The evaluation results are promising. In total, it takes about 30 hours to process all these apps and identify high- and medium-risk apps. Once this process finishes, the first-order risk analysis module reveals 2461 suspicious apps and the second-order risk analysis reports 840 apps. In total, there are 3301 suspicious apps (of which 3281 are unique – as some apps may be flagged by both risk analyses). When such an app is reported, our system also automatically generates the related execution paths that may lead to security risks. With these detailed execution paths, it took a single co-author two more days to examine these apps. In other words, our system significantly reduces the processing time of these two months' worth of apps to less than four days! We believe these results show that RiskRanker can scale to handle the current rate at which new apps are being submitted to the various Android markets.

More importantly, among these 3281 unique suspicious apps, we successfully uncover 322 (or 9.81%) zero-day malware samples[4] that belong to 11 distinct families. (The first- and second-order risk analyses contribute to identifying 40 and 282 zero-day malware instances, respectively.) In addition, from the same dataset, we also identify a further 396 (12.07%) malware samples from 18 known malware families. As a result, from our two-month dataset's apps, RiskRanker successfully detects 718 (21.88% of 3281 suspicious apps) malware samples representing 29 different families.

In summary, this paper makes the following contributions:

- To uncover zero-day Android malware in existing Android markets, we propose a proactive scheme (with two-order risk analysis) to assess the security risks introduced by these apps. To the best of our knowledge, RiskRanker is the first system that performs such a large-scale security risk analysis for zero-day malware detection.

- We have implemented a RiskRanker prototype and used it to examine 118,318 apps collected in a two-month period – September and October 2011 – from multiple Android markets. Using RiskRanker, processing this large number of apps takes less then four days. Among 3281 suspicious apps it reports, we find 718 malicious apps (from 29 malware families), 322 of them being zero-day (in 11 different families). These findings demonstrate the scalability and effectiveness of our scheme.

The rest of this paper is organized as follows: we first describe our system design in Section 2 and then present our prototyping and evaluation results in Section 3. After that, we discuss possible limitations and further improvements in Section 4. Finally, we discuss related work in Section 5 and conclude our work in Section 6.

2. DESIGN

RiskRanker is designed to scalably and accurately sift through a large number of apps from existing Android markets to uncover zero-day malware. By assessing and prioritizing potential risks from these untrusted apps, we aim to use the system to significantly narrow down the search space to a manageable size, which naturally raises the challenging requirements of *scalability*, *efficiency*, and *accuracy*. More specifically, our system must scale to handle hundreds of thousands of apps in a timely and resource-efficient manner. The system also needs to be efficient to winnow its input down to a list that is short enough to verify manually, and accurate enough to not miss malicious apps.

Figure 1 shows the overall architecture of RiskRanker. To meet the above design goals, we assess and translate

[2]Because of that, we in this paper mainly focus on the first two categories, i.e., *high-risk* and *medium-risk*.

[3]In this paper, we consider apps that have the same SHA1 value to be identical.

[4]In this paper, we consider a malicious app to be zero-day if it has not been reported before and cannot be detected by anti-virus software at the time of discovery.

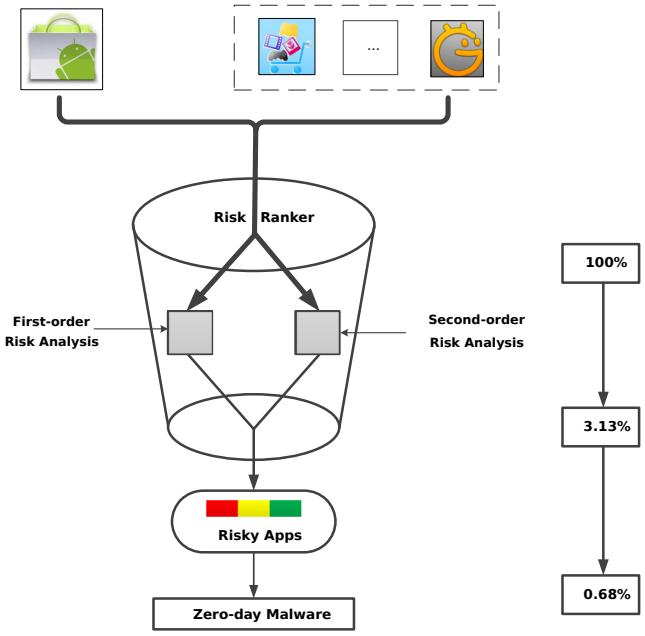

Figure 1: The RiskRanker architecture

potential security risks into corresponding detection modules of two orders of complexity. In particular, we analyze each app from an Android market with a set of analysis modules designed to detect behaviors that we classify as high- or medium-risk: The *first-order* modules handle non-obfuscated apps by evaluating the risks in a straightforward manner; The *second-order* modules capture certain behaviors (e.g., encryption and dynamic code loading) that are in themselves not of concern, but that in conjunction with others may form malicious patterns and be instrumental to detect stealthy malware.

These analysis modules ultimately produce output that includes a severity rating and related evidence to verify the behavioral pattern in each reported app. This output is then sorted by severity to produce a prioritized list of suspicious apps that merit further analysis. In the remainder of this section, we will detail these analysis modules.

2.1 First-Order Analysis

The first-order analysis modules are designed to scalably sift through untrusted apps and expose those high- and medium-risk apps.

Detecting high-risk apps RiskRanker flags any app as high-risk if it carries attack code that will exploit platform-level vulnerabilities in the OS kernel or privileged daemons to obtain superuser privileges. In Android, normal apps are typically constrained by the Linux process boundary. They are allowed to communicate with each other and may choose to contain native code (for improved performance with native speed execution). However, such a design also exposes the underlying OS kernel as well as other privileged daemons to malicious exploitation. If successful, such exploitation will allow an untrusted app to completely bypass the built-in security mechanisms, allowing it unfettered access to the device. Therefore, these exploits pose one of the greatest threats to users.

In order to detect these platform-level exploits, we distill each known vulnerability into a corresponding vulnerability-specific signature [52] to capture its essential characteristics, which will be exhibited when the vulnerability is being exploited. For example, `Exploid` [8] takes advantage of a vulnerability in the privileged `init` daemon by not verifying the source of an incoming `NETLINK` message (which contains commands for execution). Accordingly, we extract the following two preconditions as its signature: (1) it sends a message via the socket interface; (2) the message is in a particular format such as "`ACTION=add%cDEVPATH=`", which is used to trigger the vulnerability.

Table 1: An overview of existing platform-level exploits in Android

Exploit Name	Vulnerable Program	Malware with the Exploit
`Asroot` [9]	`Linux Kernel`	`Asroot`
`Exploid` [8]	`init`	`DroidDream, zHash DroidKungFu`
`GingerBreak` [29]	`vold`	`GingerMaster`
`KillingInThe NameOf` [2]	`ashmem`	-
`RATC` [12] `Zimperlich` [30]	`adbd zygote`	`DroidDream BaseBridge DroidKungFu DroidDeluxe DroidCoupon`
`zergRush` [20]	`libsysutils`	-

In Table 1, we show the list of known platform-level exploits in Android as well as the representative malware families that make use of these exploits. Notice that if the `RATC` exploit is launched by an app, it actually exploits a bug in `Zygote` instead of the `adbd` daemon. In this case, it behaves like the `Zimperlich` exploit, so we use `RATC` to represent both of these exploits.

To reduce the number of apps for exploit detection, we pre-process each app to detect the presence of native code. For each occurrence, we verify it with these signatures to spot the presence of any of these root exploits. If present, we flag the app as high-risk and take a note of the related vulnerability. Among our dataset of 104,874 distinct apps, we found that 9.42% contain native code.

Detecting medium-risk apps RiskRanker reports as medium-risk those behaviors that could result in the user being charged money surreptitiously or that upload undeniably private information to a remote server. For example, Android has a group of permissions named `android.permission-group.COST_MONEY`, which is defined as "permissions that can be used to make the user spend money without their direct involvement." [17] If abused, these features can be used to construct malware such as SMS Trojans, which send text messages to premium phone numbers that result in charges being placed on the user's phone bill. Such malware is popular due to the direct return it provides to malware authors, but these same features do have legitimate uses, typically in the form of instant-messaging, reminder, or social-networking apps.

To distinguish between such legitimate and malicious uses of potentially costly functions, we leverage the associated semantics defined in Android. Specifically, Android apps make extensive use of callbacks, and each such callback is invoked by the framework under well-defined conditions. As an ex-

ample, a button on the screen, when pressed by the user, ultimately will lead to `onClick(...)` callback. Malware that intends to charge the user without their knowledge is unlikely to do so via such a callback handler – as such handlers are triggered by user interaction. Similarly, malware that transmits sensitive data is unlikely to prompt the user before doing so. Accordingly, in our search for risky behaviors, we look for code paths that can cost the user money *without* implying such user interaction.

To find such code paths, we perform static analysis on the reverse engineered Dalvik bytecode contained in each app. In particular, we elect to use a conservative program analysis, based on symbolic execution, to determine which callbacks can result in a call to a method of interest. This process involves both control- and data-flow analysis techniques in a bid to unambiguously identify the code path. While data-flow analysis presents some challenges on Android, the properties of Dalvik's bytecode assist us with the control-flow portion of our work.

To elaborate, Dalvik (similar to Java) requires that the control-flow graph of an app should be reliably determined in advance. When Dalvik code is loaded for execution, a static verifier is run to ensure that all references in the code can be resolved. To facilitate this process, the bytecode itself does not contain operations that can lead to ambiguity: for example, there are no computed jumps or field accesses via memory addresses. At any given point in a Dalvik program's execution, the type of all its registers are known and there are a limited number of subsequent instructions that can be branched to. Furthermore, all method and field references are by name, and method execution must always start at the first instruction in the method.

We combine these factors to remove much of the ambiguity that makes the static analysis of Dalvik bytecode potentially unreliable. However, the requirements of the static verifier and our RiskRanker system only overlap but so far. Specifically, the Dalvik's static verifier only aims to ensure that the code being loaded is well-formed, and its notions of well-formedness concern only methods and classes. The verifier therefore only is designed to perform intra-method dataflow analysis, to determine the type of each virtual register at each point in the program. However, Android apps have a very complicated lifecycle, one that often features concurrency and does not require that entry points be called in any strict sequence. Given the sheer number of developers and developer styles any large-scale static analysis system will encounter, inter-method dataflow analysis will necessarily introduce ambiguity for RiskRanker. This ambiguity presents four challenges that merit further discussion.

The first source of control-flow graph ambiguity involves the class hierarchy. When a method signature specifies that a parameter is an object, this can lead to unnecessary edges from that method in the call graph. For example, all objects must ultimately descend from the `java.lang.Object` class, which has a `toString()` method. Consider what happens when a method takes a `java.lang.Object` as a parameter, and subsequently calls that parameter's `toString()` method: the method call could resolve to the `toString()` method of literally *any* class! To mitigate this problem, we employ a form of points-to analysis. Type information within a method is guaranteed to be consistent by the static verifier; therefore, when considering a potential code path, we propagate this intra-method type information across method invocations, thereby reducing the set of methods such an ambiguous method call could resolve to.

Algorithm 1: Android app analysis

Input: method call of interest, set of entry points
Output: code path, constraints

states = (method call site, no constraints)
foreach *state* ∈ *states* **do**
  remove *state* from *states*
  foreach *predecessor of state* **do**
    if *predecessor is an entry point of interest* **then**
      report the code path and constraint information
    if *predecessor is a branch comparison* **then**
      states+ = (*predecessor*, *existing constraints* + *comparison constraints*)
    else
      states+ = (*predecessor*, *existing constraints*)

Second, the possible values that each term in a comparison can take must be known to determine which paths through a program are feasible. Since the verifier does not check this, it is perfectly legal to compile an Android app that has large tracts of dead (unreachable) code, so long as that code is composed of legal methods and classes. In RiskRanker, we address this issue by treating the comparisons encountered along a code path as constraints; the problem then becomes one of constraint satisfaction, where feasible paths have satisfiable constraints. Note that concurrency can still lead to certain ambiguity, as values tested in a comparison may be defined in other threads. As these values are not defined along the current code path, they result in satisfiable constraints if any consistent value exists that would result in the correct branches being taken along the code path. However, it is possible that this consistent value may not exist in practice, resulting in some false positives. We note that this problem is, in the general case, not decidable.

In addition, there are two last potential sources of ambiguity: native code and the reflection language feature. Fortunately, both can be flagged easily, as their APIs and usages are well-defined. Native code is obviously of interest to other analysis modules already, and so is itself a cause for concern. Reflection, on the other hand, simply allows for a programmer to bypass the static verifier in order to call a method by name – where that method's name is given as a string object, rather than in bytecode. Fortunately, this confluence of the data-flow and code-flow graphs of an app is only problematic if the arguments originate outside the current method body (recall that data-flow within a method is well-defined). Therefore, it is possible to ignore a large number of reflection calls, as many such calls use constant arguments in practice. Reflection calls that rely on data defined elsewhere in the current code path can be resolved similarly by stitching together the information that is known from the intra-method dataflow analysis performed in the course of static verification. Unfortunately, it is possible that some data used for reflection originates outside the scope of the current thread. In this case, we can again employ points-to

analysis to determine the set of possible values this data can take, and thus the set of methods or fields the reflection call could be referencing.

Mindful of the challenges outlined above, we construct the control-flow graph, locate each method call of interest (e.g., those can be potentially abused for background sending of SMS messages), and perform reachability analysis. Our approach is summarized in Algorithm 1: For each method call of interest, we subsequently traverse the control-flow graph in reverse, looking for a callback method that does not imply user interaction. If such a method is found, we report the call sequence that leads to the potentially risky method call, which can then be verified. Due to the stability of these method call chains, we can further use white-listing to eliminate known-safe paths from repeated analysis (i.e., those paths that arise from a commonly-included library need only be verified once).

Our experience indicates that this algorithm is generic and can be readily extended to handle information leaks that disclose information that directly concerns the user's identity, such as their recent calls or the list of accounts. In particular, the general approach we take essentially employs *backwards slicing* to determine where the arguments to a network call originated from. If this slicing leads to a method call that obtains personal information, we flag the app as having a potential information leak and report the corresponding execution path. In other words, we essentially treat the propagation of dangerous information as a form of the type-resolution problem we addressed earlier. Intra-method dataflow is unambiguous on Android, and inter-method dataflow can be pieced together by propagating interesting data through the call chain. However, the type of fields used to store sensitive information must also be inferred, and such fields might be defined in other threads. We solve it by applying *forward slicing* to see what fields depend on the data returned by a sensitive call. This field information is then retained and referenced by the backwards slicing process used to determine where the arguments to a network operation originate from.

2.2 Second-Order Analysis

Our first-order analysis modules are mainly designed to handle non-obfuscated apps and may fall short for stealthy malware that intensively encrypts or dynamically changes its payload. To mitigate these weaknesses, we accordingly develop second-order analysis modules to collect and correlate various signs or patterns of behavior common among malware yet not among legitimate apps.

Pre-processing The first step in our second-order analysis is to capture certain distinct behavior which may not be malicious in itself but is commonly abused by existing malware. One example is the inclusion of a secondary (child) app inside a host app. The secondary app typically exists as an internal `.apk` or `.jar` file containing its executable code and is saved in the host app's `assets` or `res` directory that can be programmatically accessed. Note that the inclusion of these secondary apps is not inherently suspicious. In fact, it is a common practice that some popular ad and mobile payment service providers require the host apps to embed their ad and mobile payment frameworks inside. However, if the host app is malicious, the same mechanism can be leveraged to dynamically load and execute the embedded (malicious) child app. Motivations for this behavior are many. The child app can request additional privileges or persist after the host app has been uninstalled. Another example is that many legitimate apps use the Java encryption APIs to encrypt their communications and data. However, these same methods can be misused by malware to encrypt their malicious native code, frustrating our high-risk analysis module's efforts to detect embedded root exploits.

To recognize these abusable behaviors, we collect and aggregate various information for our second-order analysis. In our current prototype, RiskRanker is designed to automatically collect the following information: (1) the location of the secondary app; (2) background dynamic code loading and related execution path(s); (3) programmed access to internal `assets/res` directories; (4) use of encryption and decryption methods; and (5) native code execution (e.g., `Runtime.exec(...)`) and JNI accesses.

Encrypted native code execution The detection of high-risk apps in our first-order analysis sought to detect the presence of platform-level exploits in an app by using a set of signatures we developed. However, such an approach may be thwarted if malware authors attempt to encrypt the exploit code. In fact, many known Android malware store their exploit code in encrypted form within the `assets` or `res` directories, from which this encrypted code is read, decrypted and executed at runtime. Fortunately, these related behaviors are collected during our pre-processing phrase and can be correlated for their detection.

Specifically, in a normal scenario, native binaries are supposed to be contained within an app's `lib` directory. The `assets` and `res` directories are designed to contain art assets, user-interface descriptions and the like, but can also contain arbitrary data. The Android framework provides two classes, `android.content.res.AssetManager` and `android.content.res.Resources`, that gate access to this data. While many accesses to such data are innocent enough, accessing these systems in a code path that also contains the encryption and execution methods should raise red flags, as storing native binaries in such non-standard circumstances may signal that the app has something to hide.

In addition, Android contains the Java `javax.crypto` package, which provides a number of encryption and hashing functions in an easy-to-use, standardized form. While we recognize that malware could use their own decryption methods, or bundle third-party crypto libraries, the convenience this package provides appears alluring to malware authors. Consequently, in our current prototype, we opt to check for the use of its APIs.[5]

Moreover, in possible execution paths, we also look for related JNI calls as well as other methods (e.g., `Runtime.exec(...)`) to invoke native code. This is based on the observation that after the encrypted native binaries have been loaded and decrypted, they will be executed.

In RiskRanker, we combine the above three pieces of information together to detect encrypted native code execution. If an app programmatically accesses the `assets` or `res` directories, the encryption APIs, and calls `Runtime.exec(...)` in one execution path, then we assume that the app is executing an encrypted native binary stored in the `assets` or `res` directories. We classify such apps as potentially high-risk, just as with the earlier high-risk app detection in our first-

[5]A more sophisticated system could employ heuristics to detect the presence of in-app cryptographic methods [53].

Table 2: Overall results from RiskRanker

Family	# Samples	Zero-day?	First-Order Risk Analysis		Second-Order Risk Analysis	
			High-Risk	Medium-Risk	Encrypted Native Code Execution	Unsafe Dalvik Code Loading
Androidbox	13			√		
AnserverBot	185	√		√		√
BaseBridge	7		√			
BeanBot	6	√		√		
CoinPirate	1			√		
DogWars	1			√		
DroidCoupon	1	√	√			
DroidDream	2		√			
DroidDreamLight	30			√		
DroidFun	1	√		√		
DroidKungFu1	3				√	
DroidKungFu2	1		√		√	
DroidKungFu3	213				√	
DroidKungFu4	96	√			√	
DroidKungFuSapp	2	√			√	
DroidLive	10	√		√		
DroidStop	7	√		√		
FakePlayer	1			√		
Fjcon	9	√		√		
Geinimi	24		√	√		
GingerMaster	2		√			
GoldDream	21			√		
Kmin	48			√		
Pjapps	17			√		
Pushme	1			√		
RogueLemon	2	√		√		
RuPaidMarket	3			√		
TigerBot	3	√		√		
YZHC	8			√		
Total	718	11	6	20	5	1

order analysis. This module effectively leads to the discovery of 315 malware samples in five malware families, including two zero-day malware families (Section 3).

Unsafe Dalvik code loading Our next second-order detection module captures the unsafe behavior of dynamically loading untrusted Dalvik code. Typically, the bytecode that runs in a Dalvik virtual machine is from the `classes.dex` file embedded in the app or the framework itself. For extensibility, Android provides a mechanism that can be used to load and execute bytecode from an arbitrary source at runtime. Specifically, an app can leverage the `DexClassLoader` feature [11] to load classes from embedded `.jar` and `.apk` files.

The dynamic loading of new class files could potentially change the code to run and thus reduce the effectiveness of our earlier static-analysis efforts (as our analysis does not have access to all of the code that is about to run). On the other hand, we cannot consider all apps with dynamic loading behavior malicious, because this behavior can be used legitimately. For example, apps can use this mechanism to update their functionality without reinstalling the app itself. In fact, among the 118,318 apps we analyzed, 3.90% of them are using `DexClassLoader`. In our prototype, we further combine the dynamic code loading behavior with the existence of a secondary package. This can significantly reduce the number of apps for further analysis to only 424 apps. This combination leads to the discovery of the zero-day `AnserverBot` [21] malware, which accounts for 184 distinct samples.

3. PROTOTYPING & EVALUATION

We have implemented our RiskRanker prototype in Linux using Java and Python. The first- and second-order risk analysis modules are mainly implemented in Python, except for the medium-risk detection module, which is written in Java. In total, there are $3.6K$ lines of Python and $8.7K$ of Java code.

Our current prototype contains a high-risk root exploit detection module that is sensitive to seven kinds of known root exploits (Table 1). It also contains a medium-risk detection module that scans for four different kinds of behavior: sending background SMS messages, making background phone calls, uploading call logs and uploading received SMS messages. Of the medium-risk metrics, we find that background SMS sending behavior is the most useful, and will principally discuss results arising from it.[6]

[6]It is important to note that some malware samples such as `GoldDream` were only detected due to other behaviors (e.g., uploading SMS messages to a remote server).

Table 3: Malware discovery in high-risk apps

	Apps with native code	High-risk apps	Actual malware
# of apps	9877	24	14
Percentage	9.42%	0.02%	0.01%

In our prototype, to quickly index and look up various information associated with the collected apps, we use a MySQL database. In particular, before running our system, we will pre-process each app and extract related information into the database (e.g., where the app was downloaded from, whether it contains native code, etc.).

To demonstrate the effectiveness and accuracy of our prototype, we collected 118,318 apps over a two-month period – September and October 2011 – from 15 different Android markets, including the official one and 14 other alternative markets. As the same app may be present in multiple Android markets, we have in total 104,874 distinct apps: 52,208 (49.78%) of them came from the official Android Market and the remaining 52,666 (50.22%) from other marketplaces. Our prototype was deployed on a local cluster of 5 nodes, each containing 8 cores and $8GB$ of memory. The actual run of our system shows that it can handle approximately 3500 apps per hour, which means that it took about 30 hours to process all of the apps in our sample.

Among the collected 104,874 distinct apps, RiskRanker successfully uncovers 718 malicious apps in 29 malware families, including 322 zero-day malware that belong to 11 distinct new families. In Table 2, we show a detailed breakdown of our results. Specifically, the first-order risk analysis reveals 220 malware samples from 25 families. The second-order risk analysis leads to 6 malware families with 499 samples. One malware family (`DroidKungFu2`) has a sample detected by both methods, while another (`AnserverBot`) has its host app detected by the second-order techniques and its child app by the first-order techniques. These results show the effectiveness of RiskRanker.

In the following, we examine each risk analysis module and present related results.

3.1 First-Order Analysis

Detecting high-risk apps As mentioned earlier, we consider apps with root exploits to be high-risk apps. In order to detect them, we first locate all apps with native code and then compare their native code with the signatures of known root exploits. More specifically, we check the properties of each file in the app to determine whether it is an `ELF` binary. If so, we then scan this `ELF` file with our predefined root exploit signatures. If there is a match, our system reports it for manual verification.

In Table 3, we report the detection results of high-risk apps. Overall, 9877 (or 9.42%) of collected apps contain native code. Among them, we find that 24 embed root exploits. So far, we only observe three exploits being used in our dataset: `Exploid`, `RATC` and `GingerBreak`, with `RATC` being the most popular – it accounts for more than 60% of the detected samples.

Further analysis shows that among these 24 high-risk apps, 14 are actually malware from 6 distinct families. Figure 2 shows the breakdown of samples from each of these malware families: 50% of the detected malware samples are

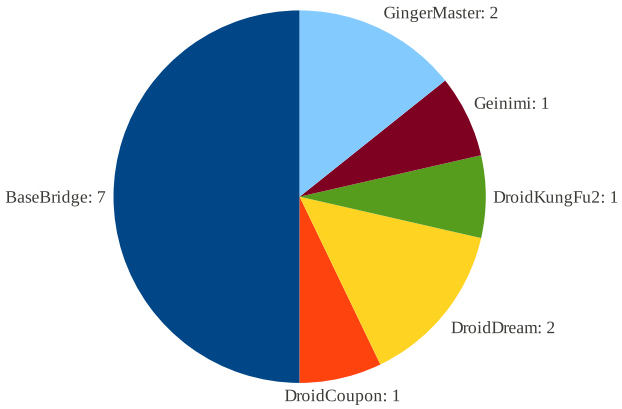

Figure 2: The breakdown of detected malware with root exploits

`BaseBridge`. From the results, we also successfully uncover two zero-day malware, i.e., `GingerMaster` [14] and `DroidCoupon` [23].

In particular, `GingerMaster` is the first malware discovered in the wild that makes use of a root exploit capable of rooting Android 2.3 devices. Similar to many others we detected in this study, `GingerMaster` repackaged its exploit code into a popular legitimate app (in this case, one that displays photographs of models). The embedded exploit code, once installed, will be triggered to root the phone and subsequently leverage the elevated privilege to download and install other apps from a remote server without the user's knowledge. The `DroidCoupon` malware similarly piggybacks on popular coupon apps by enclosing the `RATC` root exploit. If successful, it will also attempt to fetch additional apps from a remote server and install them silently. To help conceal their natures, both `GingerMaster` and `DroidCoupon` employ obfuscation techniques. For example, the root exploits used in these two malware families are saved as picture files, having `.png` as the filename suffix. `DroidCoupon` also disguises command strings and URLs as integer arrays.

Interestingly, our system spots one `Geinimi` [22] sample, which "normally" does not have a root exploit in its payload. In this case, the `Geinimi` sample is a repackaged version of `com.corner23.android.universalandroot`, a legitimate jail-breaking app with root exploits. Because of that, it is also detected by our system.

Besides these 14 malware, there are 10 additional apps that appear to have legitimate reasons to contain a root exploit: for example, `z4root` (4 samples) advertises itself as a one-click solution to root devices, while `universalandroot` (2 samples), `itfunzsupertools` (2 samples) and `holentech` (1 sample) provide very similar functionality. The last one is a security app, `com.lbe.security` (`SHA1: 34d90bbed455b370349 0a606e71d0232894c1ca4`), which contains the `GingerBreak` root exploit. In this particular case, we cannot determine whether this app should be considered malicious. Our sample contains several security apps, however, and none of the others required such functionality, so this practice is certainly unusual.

Medium-risk apps Medium-risk app detection involves constructing the whole program function call graph and then checking whether any reachable paths exist between a set of source methods and a set of sink methods. For background SMS sending behavior, we use `sendTextMessage/sendMultipart TextMessage(...)` in class `SmsManager` as sink methods. With

Table 4: Malware discovery in medium-risk apps

Family	# Samples	Zero-day?
Androidbox	13	
AnserverBot	1	√
BeanBot	6	√
CoinPirate	1	
DogWars	1	
DroidDreamLight	30	
DroidFun	1	√
DroidLive	10	√
DroidStop	7	√
FakePlayer	1	
Fjcon	9	√
Geinimi	23	
GoldDream	21	
Kmin	48	
Pjapps	17	
Pushme	1	
RogueLemon	2	√
RuPaidMarket	3	
TigerBot	3	√
YZHC	8	
Total	206	8

respect to source methods, we use a comprehensive list of entry points and callback methods that do not involve user interaction. If we find such a path, we then mark the app as a medium-risk and save the detected call chain path in the MySQL database.

In total, our system reports 2437 apps that exhibit background SMS sending behavior. Because there can be multiple paths from source to sink methods in one app, the total number of paths (3406) is larger than the number of apps. Also, while it is possible to go through these 2437 apps one by one, we find that we can still significantly reduce the number of apps that need to be analyzed. In particular, instead of analyzing each app, we opt to analyze each unique path our system detects (as such paths represent the actual behavior of concern). Moreover, as the same path can exist in multiple apps, the number of unique paths is less than the number of apps. For example, the path from the `run()` method of the `com.GoldDream.zj.zjService$1` class to the `sendTextMessage(...)` method exists in 5 different apps, all of which belong to the `GoldDream` malware family. In total, of these 3406 paths, only 1223 paths are unique.

Subsequently, we analyzed these 1223 distinct paths individually. Our experience indicates that this analysis can be done by one person under two days. When a potentially-malicious path was identified by this analyst, we forwarded it to another for confirmation. Ultimately, we discovered 94 distinct malicious paths. We then marked all the related apps corresponding to these malicious paths as malware.

Overall, we discovered 206 infected apps among 2437 medium-risk apps, representing 20 families, including 8 zero-day malware families. We tabulate these results in Table 4. The discovery of new samples from known malware families indicate that some of these families are still actively spreading in the wild. For example, the `Pjapps` malware was first discovered in February 2011. More than half a year later, it is still present in some alternative marketplaces.

Table 5: Malware discovery in second-order risk analysis

Family	# Samples	Zero-day?
DroidKungFu1 [26]	3	
DroidKungFu2 [25]	1	
DroidKungFu3 [24]	213	
DroidKungFu4 [16]	96	√
DroidKungFuSapp [18]	2	√
AnserverBot [21]	184	√
Total	499	3

Of these new discoveries, a few merit special mention. For example, the `AnserverBot` sample we detected is actually its child app, which contains the background SMS sending behavior. The `DroidStop` malware performs its activities at an interesting point in its host app's lifecycle. When the app is no longer visible to the user, DroidStop sends a SMS to a particular premium-rate number (2623588217) inside the USA. This last malware was once published on the official Android Market, but has now been removed by Google.

3.2 Second-Order Analysis

In our second-order risk analysis, we combine various information to locate apps that feature encrypted native code and unsafe Dalvik code loading behaviors.

Encrypted native code execution As discussed before, we combine three pieces of information together to detect apps that execute encrypted native binaries in the background. In order to collect this information, for each app, we first get all the paths from the app's entry points to the methods that represent the constituent behaviors that make up this greater pattern. More specifically, these methods include decryption methods (from the `javax.crypto` package), methods that access the assets or resources directories, and the `Runtime.exec(...)` method. The paths we detect to each method of interest are saved into a MySQL database. We then examine the entry point for each path in a given app; if all three types of path exist that start from the same entry point, we mark the app as potentially risky.

We found that 5495 apps contain all three types of path. However, only 328 of those apps contain all three paths originating from the same entry method. We then analyzed these 328 apps and discovered 315 malware samples from 5 different malware families, including 2 zero day malware families; the remaining 13 apps are benign. In Table 5, we show the malware reported by our second-order risk analysis of encrypted native code execution (with the exception of `AnserverBot` which was detected due to unsafe Dalvik code loading). The results are encouraging as all known `DroidKungFu` variants are successfully detected. By capturing the intrinsic characteristics of the `DroidKungFu` malware, we expect that RiskRanker will be able to capture new `DroidKungFu` variants in the future.

Unsafe Dalvik code loading Besides the detection of encrypted native code execution, our second-order analysis also captures the unsafe practice of dynamic Dalvik code loading. In combination with our child app detection logic, we can effectively identify those apps that could load new code at runtime. In our prototype, we first recognize all apps that are capable of dynamic Dalvik code loading (e.g., by looking for references to the `DexClassLoader` class' methods).

Table 6: The missed malware samples

Malware Name	#	Malware Name	#
ADRD	1	Gone60	2
BaseBridge	2	jSMSHider	1
DroidDream	1	BatteryDoctor	1
DroidDreamLight	2	TapSnake	1
FakeNetflix	1	-	-

Among the resulting apps, we further detect the presence of a child app. If such an app is found, we immediately flag its parent for further analysis.

In total, we found that 4257 apps feature dynamic code loading capability and 1655 apps contain a child package. Combining these two categories, we find 492 apps in common. After analyzing these apps, we found that some of the uses of `DexClassLoader` originate from embedded libraries, rather than the app itself. For example, one particular library from iBuildApp [15] uses `DexClassLoader` methods, and is contained in 135 apps. Furthermore, one ad library [4] and the Adobe AIR [3] platform extensively use the `DexClassLoader` feature. Accordingly, we choose to white-list these libraries to reduce the number of apps that need to be analyzed.

After white-listing these libraries, we found one particular app that dynamically loads its child package `anserverb.db` from the `assets` directory. Upon analyzing this app, we found it is actually zero-day malware – which we named `AnserverBot` – and accordingly released a security alert about it. It turns out that this malware is one of the most sophisticated pieces of Android malware we have discovered so far. Specifically, it aggressively employs several sophisticated techniques to evade detection and analysis, including Plankton-like [27] dynamic code loading, Java reflection-based method invocation, heavy use of code obfuscation and data encryption, self-verification of signatures, as well as run-time detection and removal of installed mobile security software. The detailed analysis can be find in our report [5]. In total, we detected 184 `AnserverBot` samples.

3.3 False Negative Measurement

Our results so far demonstrate the effectiveness of RiskRanker in uncovering new (zero-day) Android malware. In the following, we further measure the false negatives of our system. To this end, we download malware samples from the public contagio dump repository [10] and use them as the ground truth. Note that the malware samples in this repository are contributed by volunteers and our study shows that there are some duplicates. After removing these duplicates, there are 133 distinct samples from 31 different families.

Our system reports 121 of the 133 samples we collected from the contagio repository. The remaining 12 malware samples are summarized in Table 6. Upon analysis, we discover that the samples that our system fails to detect all fall outside the scope of our current analysis in one of three ways. First off, some malware families engage in malicious behaviors that our system does not attempt to detect. For example, `FakeNetflix` impersonates a Netflix app in a bid to collect the user's account and password information; such social-engineering attacks are very difficult for an automated system to distinguish from legitimate app functionality. `BatteryDoctor`, `Gone60`, and `TapSnake` are spyware that harvest personal information and human judgment is still necessary to distinguish them from many other legitimate apps in the marketplace that may collect user information. The `ADRD` sample engages in click fraud by retrieving a list of ad URLs from a remote server and then visiting them in the background. Second, the missed samples do not share the same malicious payload as others in the same family. For example, our system missed `DroidDreamLight` samples that lack the background SMS sending behavior seen in other samples in the same family. Also, the missed `BaseBridge` samples do not contain the root exploits common to that family. Third, there are some samples that are, in a sense, guilty by association. For example, `jSMSHider` installs a malicious child app, while the missed `DroidDream` "sample" is in fact the malware's innocuous child app. Neither of these samples themselves engage in actual malicious behavior, but the apps associated with them do.

3.4 Malware Distribution Breakdown

We further analyze the distribution of the malware we detected among the Android markets we studied. Sadly, we find that malware can be detected in all of the markets we examined, including the official Android Market. In the worst case, 220 samples were detected in a single alternative market, representing 3.06% of all apps collected from that market. Four alternative markets contain more than 90 malware samples apiece. Specifically, if we sum up the malware samples from these four alternative markets, they together account for 85.98% of all malware samples we discovered. We stress that while no sizable market can be expected to be perfectly secure, some markets clearly do better than others at policing their contents. For example, in the official Android Market, only 2 apps out of $52,208$ were malicious. Although we do not know the exact reason why Google's market is so much cleaner than some of the others, we believe that it is likely due to the adoption of Google Bouncer [6] service. Unfortunately, there is limited public information on how Bouncer works.

4. DISCUSSION

While our prototype demonstrates promising results, it is still a proof-of-concept system, and therefore has several important limitations. In this section, we will examine these limitations and suggest possible areas of improvement.

To begin with, our root-exploit detection scheme depends on signatures, which obviously implies that it can detect only known exploits and may also miss encrypted or obfuscated exploits. Our second-order risk analysis alleviates the situation and at present appears to work very well. However, if such techniques start to hinder the deployment of malware in app markets, malware authors might respond by attacking some of the assumptions that make this method work so well. For example, our prototype only considers the `javax.crypto` libraries for convenient encryption detection; nothing prevents an attacker from implementing their own in-app encryption or decryption scheme. Similarly, instead of packaging their native code in the assets or resource directories, such code could be included as large constant arrays in the Dalvik binary itself. While it is possible to counter all of these strategies, it is obvious that the situation can and likely will grow more complicated as malware matures on the platform.

Similarly, our dynamic Dalvik code execution scheme has limitations. Child apps are not the only sources of Dalvik

code; such code can be downloaded from the Internet or stored as raw data contained in a constant array, for example. Nothing prevents an attacker from encrypting or otherwise obfuscating a child app, either, in much the same way that native binaries are hidden today. In the long term, these concerns may best be addressed by incorporating some dynamic analysis techniques, such as fuzzing, into systems like RiskRanker. Also, while static analysis may determine that dynamic code loading is taking place, it may not be very effective at identifying precisely what dynamic code is being loaded.

Our medium-risk modules are similarly imperfect. We test for only four distinct behaviors, for example, and our dataflow analyses do not account for code lying outside the execution path very well. While these are general concerns that can be remedied by applying more sophisticated techniques drawn from the existing program analysis literature, likely at the cost of additional processor time, we do recognize one interesting practical matter that came out of our experiences collecting the data for this work. A fraction of Android apps are obfuscated to frustrate casual analysis. When we identified 1223 unique paths using our medium-risk analyses, 59 of those paths had their class names obfuscated. Obfuscation of this sort leads to two challenges. First, slightly different samples of the same app can have wildly different reported code paths that in fact represent the same behavior, because obfuscation is generally very sensitive to small changes in the input app. For example, the entry function `com.package.Class.run()` may become `a.b.c.run()` in one sample, yet `a.c.d.run()` in another. Secondly, known-safe app paths cannot be white-listed to save analyst effort over time; a known-safe ad library contained in a large number of apps, for example, may still consume analyst-hours of effort when obfuscated apps that contain it are uploaded to a market. Obfuscation is actually recommended by Google [19], so this problem is here to stay. The same issues apply to the second-order encrypted native binary execution analysis, as well. To resolve these issues, it would be better to report paths based on their semantic meaning – what they actually *do* – rather than simply their "names." Unfortunately, obfuscation can also rearrange (in limited ways) the opcodes along an execution path, so this is not a trivial problem to solve. We leave such semantic path description techniques to future work.

Finally, it is important to note that the search for malware is not always black and white. Many risky apps threaten user security and privacy, but are not necessarily malware. Sometimes, these apps are simply written in a potentially dangerous way, but are published by a reputable company; for example, Adobe AIR uses dynamic code execution, which a more aggressive system would flag. Other apps take pains to justify potentially dangerous actions to the user. For example, in the Chinese app markets, some apps display a user agreement when they are first launched, stating that the app will use a premium-rate SMS number to bill the user on some recurring basis. Some of these apps add this functionality to repackaged versions of existing apps, and so may be considered a soft kind of malware. It is impossible to determine that such apps are in fact malware, however, when they are considered in isolation. We also see many instances of classical gray-area malware in the app markets, particularly spyware. Many apps collect more information than they need to function and blithely send it to external parties – it is simply very easy to get such information on mobile platforms. However, it is an open question on where the line should be drawn on such information leaks, and how much apps need to disclose to the user about how their information is being used.

5. RELATED WORK

Recently, smartphone security has been an area of active research. Many researchers have identified areas of concern and proposed solutions to problems on smartphone operating systems. For example, TaintDroid [38] and PiOS [37] were developed to identify information leaks on smartphone platforms, where TaintDroid used dynamic analysis techniques on Android and PiOS applied program slicing to iOS binaries. A raft of follow-up work then sought to address these information leaks by altering the frameworks involved: Apex [48], MockDroid [32], TISSA [57] and AppFence [46] all offer extensions to the Android framework to provide finer-grained control over an app's access to potentially sensitive resources. Most of these efforts are aimed at addressing this problem on the user's phone; RiskRanker, on the other hand, attempts to identify such risky behaviors at the app market. Of the systems listed previously, RiskRanker has the most in common with PiOS, in that both employ program slicing to better understand app behavior. However, PiOS targets iOS binaries for information leaks while RiskRanker looks for potential security risks (including platform-level root exploits and background SMS message-sending) for zero-day Android malware detection.

Another line of research deals with the confused-deputy problem [45] on Android, where inter-process communication channels can inadvertently expose privileged functionality to unprivileged callers. ComDroid [34] and Woodpecker [44] both employ static analysis techniques to detect such attacks in third-party and firmware apps, respectively. To deal with this problem, QUIRE [36] and Felt *et al.* [42] offer extensions to the Android IPC model that allow the ultimate implementor of a privileged feature to check the IPC call chain to ensure unprivileged apps can not launch confused-deputy attacks unnoticed. Similarly, Bugiel *et al.* [33] use a run-time monitor to regulate communications between apps. RiskRanker at present does not check for confused-deputy attacks that target the IPC layer, but its design was informed by our experiences developing the Woodpecker system. We consider these efforts to be complementary to RiskRanker, because one of the lessons learned from these systems is that confused-deputy attacks do not target well-understood, standardized framework components. For example, Schrittwieser *et al.* [51] find that recent Internet-based messaging apps contain security flaws that can allow attackers to hijack accounts, spoof sender-IDs, or even enumerate subscribers. RiskRanker could be extended to identify such vulnerabilities in a similar fashion to ComDroid or Woodpecker, but it would be more difficult to automate the discovery of malware that attempts to *exploit* such vulnerabilities.

Other systems attempt to use or extend the Android permission system to defend against malware. For example, Kirin [40] blocks the installation of apps that request particularly dangerous combinations of permissions, while Saint [49] lets app developers specify permission policies that constrain permission assignment at install-time and use at run-time. RiskRanker conceptually has some similarities with these

systems, in that its second-order analyses aim to identify patterns of seemingly innocent API uses that can be indicators of malware. However, Kirin is concerned about the end user and Saint the app developer, while RiskRanker targets app markets themselves. Another interesting system in this category is Stowaway [41], which aims to identify instances of app over-privilege, where an app requests more permissions than it uses. Such over-privilege itself could be used as an input to RiskRanker in the future.

Besides the above defenses, some work has been proposed to apply common security techniques from the desktop to mobile devices. For example, L4Android [47] and Cells [31] apply virtualization techniques to the mobile space, allowing for multiple virtual cellphones to be run on one device, isolated from one another. Meanwhile, MoCFI [35] enforces control-flow integrity in iOS apps without requiring access to their source code or cooperation from their developers. These approaches naturally complement RiskRanker by providing defense-in-depth, but again are designed to be deployed on endpoint devices.

Some work has focused on malware and the overall market health. Felt *et al.* [50] surveyed 46 malware samples on three different smartphone platforms, discussing the incentives that motivated their creation and possible defenses against them. However, this work did not discuss how to discover this malware to begin with. Enck *et al.* [39] studied the 1100 top free apps from the official Android Market to understand their general security and privacy characteristics. Our work has a much stronger emphasis on malware detection than privacy leak detection. MalGenome [55] aims to systematically characterize existing Android malware from various aspects, including their installation methods, activation mechanisms as well as the nature of carried malicious payloads. Note that it did not focus on the methodology for discovering Android malware. However, the insights behind it is instrumental for RiskRanker to look for certain malicious patterns. Finally, we have developed two related systems in the past. The first, DroidRanger [56], aims to mainly detect known malware in the existing app markets. It requires malware samples to extract behavioral signatures for detection. RiskRanker, on the other hand, is designed to actively detect zero-day malware without relying on malware specimens. Also, while our second-order risk analysis shares a similar spirit with the two basic heuristics in DroidRanger, their differences are fundamentally rooted in their distinct goals: DroidRanger was designed to measure the overall health of an app market, whereas RiskRanker is designed to uncover more sophisticated zero-day malware by assessing and ranking potential risks in each app. The second related system, DroidMOSS [54], is designed to detect repackaged apps using fuzzy hashing. RiskRanker does not have any notion of whether an app has been repackaged, although many malware samples are bundled with a repackaged version of a legitimate app; likewise, DroidMOSS does not aim to detect malware. AdRisk is another recent system [43] that is proposed to systematically identify potential risks posed by existing in-app ad libraries. Although the ad libraries identified by AdRisk do not seem to have malicious intent, they contain potential risks ranging from leaking user's private information to executing untrusted code from the Internet. This potential to do harm places such libraries in a gray area.

6. CONCLUSION

In this paper, we present a proactive scheme to scalably and accurately sift through a large number of apps in existing Android markets to spot zero-day malware. Specifically, our scheme assesses the potential security risks from untrusted apps by analyzing whether dangerous behaviors are exhibited by these apps (with two-order risk analysis). We have implemented a prototype of RiskRanker and evaluate it using 118,318 apps from a variety of Android markets to demonstrate its effectiveness and accuracy: among the apps in the sample, our system successfully discovered 718 malware samples in 29 families, including 322 zero-day specimens from 11 distinct families.

Acknowledgment

We would like to thank our shepherd, Patrick McDaniel, and the anonymous reviewers for their insightful comments that greatly helped improve the presentation of this paper. We also want to thank Zhi Wang, Wu Zhou, Deepa Srinivasan, Minh Q. Tran, Chiachih Wu and Lei Wu for numerous helpful discussions.

7. REFERENCES

[1] 260,000 Android users infected with malware. http://www.infosecurity-magazine.com/view/16526/260000-android-users-infected-with-malware/.

[2] adb trickery #2. http://c-skills.blogspot.com/2011/01/adb-trickery-again.html.

[3] Adobe AIR 3. http://www.adobe.com/products/air.html.

[4] AdTouch. http://www.adtouchnetwork.com/adtouch/sdk/SDK.html.

[5] An Analysis of the AnserverBot Trojan. http://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_Analysis.pdf.

[6] Android and Security. http://googlemobile.blogspot.com/2012/02/android-and-security.html.

[7] Android Market Statistic. http://www.androlib.com/appstats.aspx.

[8] android trickery. http://c-skills.blogspot.com/2010/07/android-trickery.html.

[9] Asroot. http://milw0rm.com/sploits/android-root-20090816.tar.gz.

[10] Contagio mobile malware mini dump. http://contagiominidump.blogspot.com/.

[11] DexClassLoader. http://developer.android.com/reference/dalvik/system/DexClassLoader.html.

[12] Droid2. http://c-skills.blogspot.com/2010/08/droid2.html.

[13] Gartner Says Sales of Mobile Devices Grew 5.6 Percent in Third Quarter of 2011; Smartphone Sales Increased 42 Percent. http://www.gartner.com/it/page.jsp?id=1848514.

[14] GingerMaster: First Android Malware Utilizing a Root Exploit on Android 2.3 (Gingerbread). http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/.

[15] iBuildApp. http://ibuildapp.com/.

[16] LeNa (Legacy Native) Teardown_Lookout Mobile Security. http://blog.mylookout.com/wp-content/uploads/2011/10/LeNa-Legacy-Native-Teardown_Lookout-Mobile-Security1.pdf.

[17] Manifest.permission_group definitions. http://developer.android.com/reference/android/Manifest.permission_group.html.

[18] New DroidKungFu Variant – DroidKungFuSapp – Emerges! http://www.csc.ncsu.edu/faculty/jiang/DroidKungFuSapp/.

[19] ProGuard. http://developer.android.com/guide/developing/tools/proguard.html.

[20] Revolutionary - zergRush local root 2.2/2.3. http://forum.xda-developers.com/showthread.php?t=1296916.

[21] Security Alert: AnserverBot, New Sophisticated Android Bot Found in Alternative Android Markets. http://www.csc.ncsu.edu/faculty/jiang/AnserverBot/.

[22] Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild. http://blog.mylookout.com/2010/12/geinimi_trojan/.

[23] Security Alert: New Android Malware – DroidCoupon – Found in Alternative Android Markets. http://www.csc.ncsu.edu/faculty/jiang/DroidCoupon/.

[24] Security Alert: New DroidKungFu Variant – AGAIN! – Found in Alternative Android Markets. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu3/.

[25] Security Alert: New DroidKungFu Variants Found in Alternative Chinese Android Markets. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu2/.

[26] Security Alert: New Sophisticated Android Malware DroidKungFu Found in Alternative Chinese App Markets. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu.html.

[27] Security Alert: New Stealthy Android Spyware – Plankton – Found in Official Android Market. http://www.csc.ncsu.edu/faculty/jiang/Plankton/.

[28] Smartphone shipments tripled since '08. Dumb phones are flat. http://tech.fortune.cnn.com/2011/11/01/smartphone-shipments-tripled-since-08-dumb-phones-are-flat/.

[29] yummy yummy, GingerBreak! http://c-skills.blogspot.com/2011/04/yummy-yummy-gingerbreak.html.

[30] Zimperlich sources. http://c-skills.blogspot.com/2011/02/zimperlich-sources.html.

[31] Andrus, J., Dall, C., Van't Hof, A., Laadan, O., and Nieh, J. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (2011), SOSP '11.

[32] Beresford, A. R., Rice, A., Skehin, N., and Sohan, R. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th International Workshop on Mobile Computing System and Applications* (2011), HotMobile '11.

[33] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.-R., and Shastry, B. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security* (2012), NDSS '12.

[34] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services* (2011), MobiSys 2011.

[35] Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nurnberger, S., and Sadeghi, A.-R. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security* (2012), NDSS '12.

[36] Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. S. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium* (2011), USENIX Security '11.

[37] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th Annual Symposium on Network and Distributed System Security* (2011), NDSS '11.

[38] Enck, W., Gilbert, P., Chun, B.-g., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation* (2010), USENIX OSDI '10.

[39] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium* (2011), USENIX Security '11.

[40] Enck, W., Ongtang, M., and McDaniel, P. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS '09.

[41] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), CCS '11.

[42] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Security Symposium* (2011), USENIX Security '11.

[43] Grace, M., Zhou, W., Jiang, X., and Sadeghi, A.-R. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks* (2012), ACM WiSec '12.

[44] Grace, M., Zhou, Y., Wang, Z., and Jiang, X. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security* (2012), NDSS '12.

[45] Hardy, N. The Confused Deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review 22* (October 1998).

[46] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (2011), CCS '11.

[47] Lange, M., Liebergeld, S., Lackorzynski, A.,

Warg, A., and Peter, M. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices* (2011), CCS-SPSM '11.

[48] Nauman, M., Khan, S., and Zhang, X. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security* (2010), ASIACCS '10.

[49] Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 2009 Annual Computer Security Applications Conference* (2009), ACSAC '09.

[50] Porter Felt, A., Finifter, M., Chin, E., Hanna, S., and Wagner, D. A Survey of Mobile Malware In The Wild. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices* (2011), CCS-SPSM '11.

[51] Schrittwieser, S., Fruhwirt, P., Kieseberg, P., Leithner, M., Mulazzani, M., Huber, M., and Weippl, E. Guess Who's Texting You? Evaluating the Security of Smartphone Messaging Applications. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security* (2012), NDSS '12.

[52] Wang, H. J., Guo, C., Simon, D. R., and Zugenmaier, A. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM 2004 Conference* (2004), ACM SIGCOMM '04.

[53] Wang, Z., Jiang, X., Cui, W., Wang, X., and Grace, M. ReFormat: Automatic Reverse Engineering of Encrypted Messages. In *Proceedings of the 14th European Symposium on Research in Computer Security* (September 2009), ESORICS '09.

[54] Zhou, W., Zhou, Y., Jiang, X., and Ning, P. DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy* (2012), CODASPY '12.

[55] Zhou, Y., and Jiang, X. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy* (2012), IEEE Oakland '12.

[56] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security* (2012), NDSS '12.

[57] Zhou, Y., Zhang, X., Jiang, X., and Freeh, V. W. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing* (2011), TRUST '11.